Active Tags for Semantic Analysis

Ivan Habernal and Miloslav Konopík

University of West Bohemia, Department of Computer Sciences,
Univerzitní 22, CZ - 306 14 Plzeň, Czech Republic
{habernal,konopik}@kiv.zcu.cz

Abstract. We propose a new method for semantic analysis. The method is based on handwritten context-free grammars enriched with semantic tags. Associating the rules of a context-free grammar with semantic tags is beneficial; however, after parsing the tags are spread across the parse tree and it is usually hard to extract the complete semantic information from it. Thus, we developed an easy-to-use and yet very powerful mechanism for tag propagation. The mechanism allows the semantic information to be easily extracted from the parse tree. The propagation mechanism is based on an idea to add propagation instruction to the semantic tags. The tags with such instructions are called *active tags* in this article. Using the proposed method we developed a useful tool for semantic parsing that we offer for free on our internet pages.

1 Introduction

Semantic analysis is a very important part of natural language processing. Its purpose is to extract and describe the meaning that is contained in a sentence. The ability to understand a sentence or an utterance is surely essential for many natural language processing tasks.

In this article we describe a method that is based upon context-free grammars (CFG) enriched with semantic tags. In this approach the grammar describes the structure of analyzed sentences and the semantic tags describe the semantic information.

The process of analyzing sentences using a grammar is called *parsing* and a result of parsing is called a *parse tree*. The parse tree describes the structure of a sentence. If we use a grammar enriched with semantic tags, the semantic information is spread across the parse tree in the semantic tags. There are several ways of extracting semantic information from the parse tree. Some of the common ways as well as the newly proposed approach will be presented in this article.

The approach described in this article relies on handwritten grammars and the semantic tags. The necessity to create the grammar rules and the tags manually could be perceived as a drawback when compared to new approaches that use stochastic algorithms for automatic learning of semantic analysis rules (e.g. [7]). However, some experiments show (e.g. [2]) that stochastic algorithms are very good at identifying wide variety of sentence types but their weakness lies

in capturing complex sentence phrases such as date/time expressions or spoken numbers expressions, residence address phrases, etc. A stochastic algorithm is usually not capable to learn such complex relations. On the contrary we show that the handwritten grammars are capable of capturing these expressions easily. Moreover our method is able to formalize the semantics thus it is not complicated to produce a uniform representation of the semantics for several structurally different but semantically uniform phrases.

The idea to propagate the information from bottom to top while parsing is not new and was used in many CFG-based approaches (e.g. [1]). However, in such approaches the grammar designer is forced to use a fixed syntax or ECMA scripts [4], etc. Such requirement could be useful for some tasks but generally it is limiting because the given syntax could be either too complicated or not expressive enough. The approach described in this article is designed to be general. The grammar designer is allowed to use a script or to create structured objects or any other formalism suitable for the particular application.

The rest of this article is organized as follows. At first we describe an approach of semantic analysis with semantic tags that do not use the proposed *active tags*. We show the drawback of this approach. Then we introduce and explain our approach with active tags. Afterwards we show some examples of parsing using this method. Finally we summarize the proposed method and give links where the software implementation of this method can be obtained.

2 Parsing with Semantic Tags

Formal grammars are a useful formalism for analyzing sentences. The idea to use formal grammars for semantic analysis was first introduced by Burton [3] in formalism called *semantic grammars*. This formalism freely mixes syntax and semantics in individual grammar rules. Semantic grammars hence combine aspects of syntax and semantics in a simple uniform framework. After parsing an input sentence with respect to a given semantic grammar, the semantics is stored in a parse tree. However, the parse tree contains all words from the sentence and many of them do not carry any semantic information.

The semantic tags help to solve two problems. Firstly, semantic tags help to express which words in the parse tree carry relevant semantic content and which words serve purely syntactic needs in a sentence. The semantically relevant words are associated with the semantic tags that represent their semantics and the words that are not semantically relevant are left without the semantic tags. Secondly, they are used to formalize the semantic information contained in the words of a sentence. E.g. the word “twenty” is formalized as “20”.

Figure 1 shows a parse tree for a sentence from a voice driven chess game. The semantic tags are placed in curly brackets and clearly describe the meaning contained in the sentence. The output semantic tags are `figure=B,col=B,row=3`.

The example in figure 1 also shows a drawback of this approach. Just imagine a sentence ”I want to go from b three to d three”. Now the obtained set of tags will be `col=B, row=3, col=d, row=3`. In such a case it is difficult to distinguish

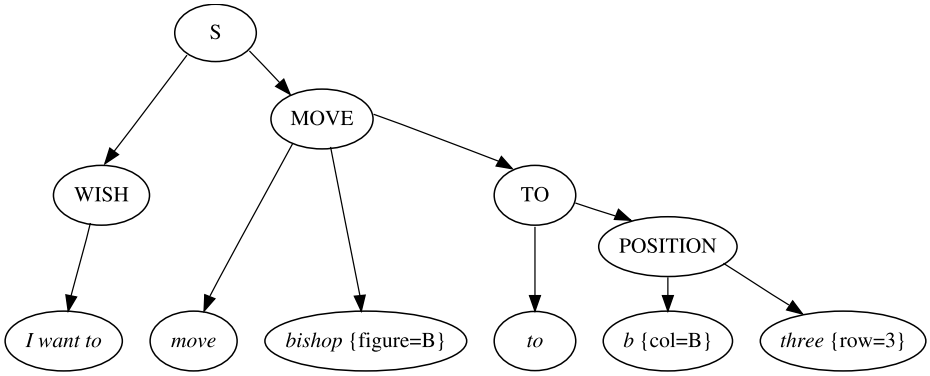

Fig. 1. Example of a parse tree with semantic tags.

what is the source position and what is the destination position. The next chapter not only shows how it is possible to elegantly solve this problem but it also shows that the semantic tags can deal with much more complicated problems.

3 Active Tags for Semantic Parsing

The semantic information in parse trees is spread across the tree in the semantic tags and the information need to be extracted. It is possible to go through the tree and store all the tags into a vector. Then it is easy to read the semantic information but the structural information is lost (see the chess example in previous section and the problem of the source and destination positions). Other possible approach is to leave the tags in the parse tree and extract the semantic information by a dedicated algorithm. In that way the structural information is not lost but this approach requires to write a special program code for each grammar. To avoid all these complications we developed a formalism of the so-called *active tags*. An active tag is a semantic tag enriched with a special processing instruction that controls the process of merging the pieces of semantic information in the parse tree. When the active tags are used and evaluated, the semantics is extracted from the tree in a form of one resulting tag that contains the complete semantic information.

The semantic information from the tree is joined in the following way. Each superior tag is responsible for joining the information from the tags that are placed directly below the superior tag. By a recursive evaluation of the active semantic tags, the information is propagated in a bottom-up manner in the tree.

An active tag contains at least one reference to a sub-tag. During evaluation the reference to a sub-tag is replaced with a value of the sub-tag. The reference has the following notion: #number (e.g. #2). The sub-tags are automatically numbered in the same order as stated in the grammar. Then the number in the

sub-tag reference says which tag with a given number will be used to replace the reference.

The core of our method is based on replacing the tag references with actual values of the sub-tags. Despite how simple this principle may seem it is capable of creating very complicated semantic constructs.

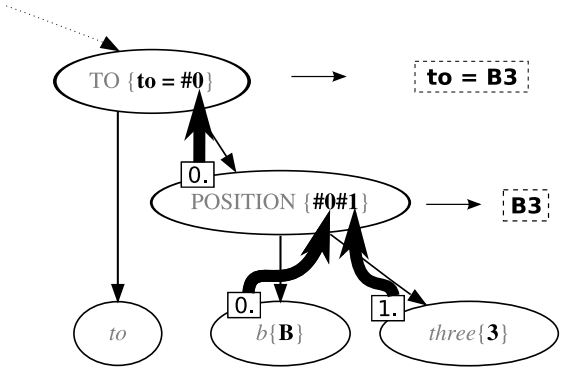

Fig. 2. A parse tree for a chess game command

For example see figure 2. It shows how simple it is to solve the chess problem from Section 2. The sub-tags (`B` and `3`) of the node `POSITION` are automatically numbered and the value of the active tag in the node `POSITION` is constructed by replacing the references `#0` and `#1` with the values `B` and `3` so the string `B3` is created. Similarly the reference `#0` in the node `TO` is replaced with the first sub-tag value and the tag `from=B3` is created. And so on.

In this way one can create scripts in the Java language, Perl scripts, ECMA scripts, attribute-value pairs and arbitrary other scripts or values. The interpretation of the resulting tag is separated from the creation of the result tag.

3.1 Spoken Numbers Analysis Example

This section shows an comprehensive example that demonstrates the processing of spoken number phrases. We reduced the grammar to numbers from 0 to 999 999 because of space limitation.

The semantic grammar with active tags is defined as follows:

```
#JSGF V1.0 UTF-8;


<S> = <thousands> {#0} | <hundreds> {#0} | <one_to_hundred> {#0};

<thousands> = (<thousand> <hundreds>) {#0+#1} | <thousand> {#0} |
```

```
              (<thousand> and <one_to_hundred>) {#0+#1};
<thousand> =  (<numeral> thousand) {#0*1000} |
              (<ten_to_thousand> thousand) {(#0)*1000};
<ten_to_thousand> = (<tens> <numeral>) {#0*10+#1} | <tens> {#0*10} |
   <teen> {#0} | <hundred> {#0} | (<hundred> <one_to_hundred>) {#0+#1};

<hundreds> = <hundred> {#0} | (<hundred> and <one_to_hundred>) {#0+#1};
<hundred> = <numeral> {#0 * 100} hundred;
<one_to_hundred> = (<tens> <numeral>) {#0*10+#1} | <tens> {#0*10} |
                   <teen> {#0} | <numeral> {#0};

<tens> = twenty {2} | thirty {3} | forty {4} | fifty {5} |
         sixty {6} | seventy {7} | eighty {8} | ninety {9};

<teen> = ten {10} | eleven {11} | twelve {12} | thirteen {13} |
         fourteen {14} | fifteen {15} | sixteen {16} |
         seventeen {17} | eighteen {18} | nineteen {19};

<numeral> = one {1} | two {2} | three {3} | four {4} | five {5} |
            six {6} | seven {7} | eight {8} | nine {9};
```

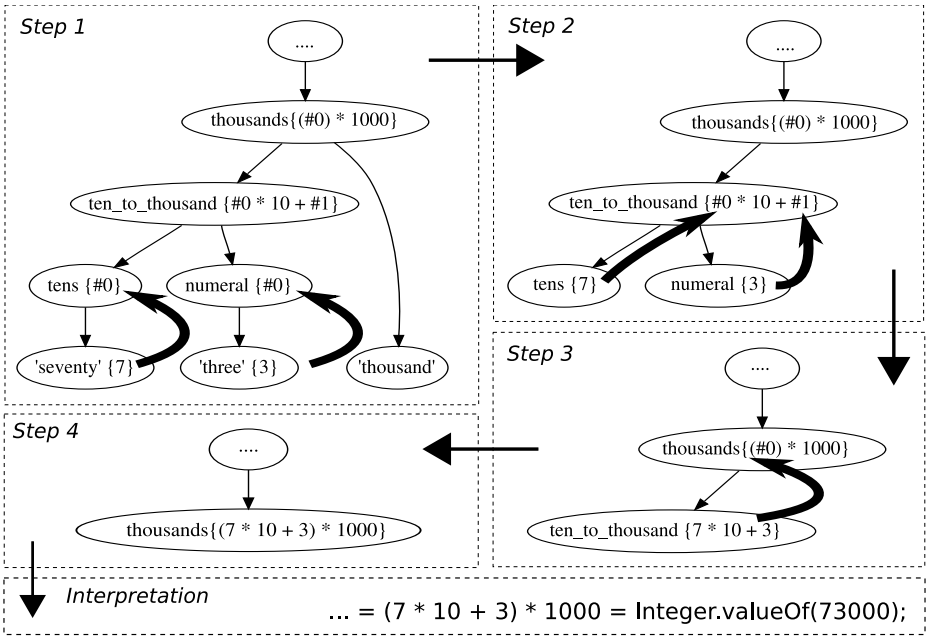

Fig. 3. An example of tag processing for spoken numbers domain

The processing of the phrase "seventy three thousand" is depicted in figure 3. During the tags extraction phase (steps 1 to 4) the resulting tag "$(7*10+3)*1000$" is created. Then it is simply interpreted (in our case with the BeanShell [5]) and the number 73000 is generated.

3.2 Formal Definition of a Grammar with Active Tags

Context-Free Grammar enriched with active semantic tags is a 6-tuple $G = (N, \Sigma, \mathcal{T}, \mathcal{V}, \mathcal{R}, S)$, where

- N is a set of non-terminal symbols (or "variables"). The non-terminals are enclosed in angle brackets (e.g. $\langle$nonterminal1$\rangle$),
- Σ is a set of terminal symbols,
- $\mathcal{T}$ is a set of semantic tags. The tags are strings enclosed in curly brackets e.g. {tag1},
- $\mathcal{V}$ is a set of active semantic tags. The active tags are like nonactive tags enclosed in curly brackets but contain at least one reference denoted by # and a number e.g. {#1 tag1 #2}
- $\mathcal{R}$ is a set of production rules, each of the from $A \rightarrow X_1Y_1\ X_2Y_2\ ...\ X_nY_n$, where $A \in N$, $X_i \in (N \cup \Sigma)$, $Y_i \in (\mathcal{T} \cup \mathcal{V} \cup \emptyset)$,
- $S \in N$ is the start symbol.

3.3 Advanced Parsing of Terminals

A classical approach to parsing assumes that the words of an input sentence exactly correspond to the terminals in a grammar. The parser thus splits the sentence into words (tokens) and tries to match the words to the appropriate rules defined in the given grammar. However there are cases when it would be useful to add some logic to the terminal matching procedure - for example when it is required to use ontology (e.g. to use hypernym as a terminal), or morphology (e.g. lemma), or to use a regular expression, etc. Our parser offers an ability to incorporate these techniques into the terminal matching procedure.

To illustrate this idea an example for languages with a rich morphology is given. For a morphologically rich language it can be very complicated to write a fully comprehensive grammar containing all possible lexeme forms. In some cases using a lemma (instead of a list of possible forms) may rapidly simplify the grammar. If the lemmatization is applied on the input sentence, the parser chooses the appropriate method for matching tags, according to the settings.

Consider a short example from a grammar generating numbers in Czech and the input sentence *"Dvacátého druhého ledna"* (twenty second January).

```
<desitky> = dvacátý {20} | dvacátá {20} | dvacátému {20} |
            dvacátého {20} | ...
```

This fraction of the grammar shows some word forms of the word "dvacet" (dvacet means twenty in Czech). In the case of this grammar a standard parser

chooses the rule generating the first word ("dvacátého") properly. The main disadvantage is that there must be an exhausting list of words covering the particular lexeme for each morphological form. Using the alternative parsing approach, we can rewrite the grammar as follows: `<desitky> = @L_dvacátý {20}`.

The `@L_` prefix means that the parser will accept the lemma. Once the input sentence is preprocessed by lemmatization, the parser obtains the following input *"Dvacátého;dvacátý druhého;druhý ledna;leden ..."*, where the second form is a lemma.

The framework also provides matching by regular expressions which can be useful for e.g. parsing text containing symbols. For instance this approach allows parsing dates in written form (11th January 2008), where the rule generating/accepting the day of month would be `<day-of-month> = @R_[1-3]?\d\th`.

4 Practical Applications

We have developed a framework for semantic analysis with active tags in the Java programming language. The grammars are in the JSGF format [6] and the software uses our own partial implementation of JSAPI[1]. The final script obtained from the parse tree evaluation is independent on a programming language; the framework has default and native support for scripting in Java language (using BeanShell [5] – a dynamic scripting language for Java platform).

The framework has been tested in various domains. The proposed method is used as a semantic parsing algorithm in a voice-driven chess game [8]. We displayed this application in many demonstrations where we encountered many variants of how to tell a chess application what move it is supposed to do. The proposed method allowed us to quickly incorporate new variants into the semantic analysis without changing the program code of the application. Another advantage is that the algorithm provides us with consistent semantic analysis. A uniform semantic representation is created for all variants with the same meaning but possibly with completely different grammar rules.

We also develop a semantic analysis algorithm for spoken queries to an internet search engine [2]. In this application the domain is much broader and it is impossible to manually write a grammar that would cover all possible questions. Thus we use a stochastic algorithm as a main semantic analysis algorithm. However, we found that we need to perform a local parsing to identify complex phrases such as date/time expressions, spoken numbers, and others. After parsing the input with the proposed semantic parser the main stochastic parser works with these phrases as atoms without the necessity to deal with them. In this task, the proposed method proved to be a very good method of local parsing.

[1] Java Speech API specifies an interface for speech processing on the Java platform.

5 Conclusion

In this article we presented a simple and easy-to-use framework for semantic information extraction. It is based on context-free grammars and a mechanism of active tags.

The main difference between our approach and some similar approaches (e.g. [1]) is that our mechanism of propagation of semantic tags is not connected with any script. This fact simplifies the formalism and also it makes the formalism more universal because the interpretation of tag values is separated from the tag propagation mechanism. It is thus possible to use arbitrary interpretation method. We also deal with the problem of parsing morphologically rich languages and the problem of parsing inputs with symbols.

We found in our experiments that this approach is extremely helpful in building semantic analysis applications. We believe that this approach can be used in many other applications that require natural language understanding. Thus we decided to release the software implementation of the proposed semantic analysis method at our internet pages. The developed software is not burdened by any licence restriction; we provide it for free under LGPL licence.

Download

The software implementation can be downloaded from https://liks.fav.zcu.cz/mediawiki/index.php/LINGVOParser.

Acknowledgements

This work was supported by grant no. 2C06009 Cot-Sewing.

References

1. A. Hunt B. Lucas, W. Walker. Ecmascript Action Tags for JSGF (proposal), Septemeber 1999. http://x-language.tripod.com/ECMAScriptActionTagsforJSGF.html.
2. Konopík, M.; Mouček, R. Towards Semantic Analysis of Spoken Queries. *In SPECOM 2007 proceedings.* Moscow, 2007. ISBN 6-7452-0110-X.
3. J. S. Brown and R. R. Burton. Multiple representations of knowledge for tutorial reasoning. In *Representation and Understanding*, pages 311–349. New York, 1975.
4. ECMA. ECMAScript language specification. ISO/IEC 16262, http://www.ecma.ch/ecma1/STAND/ecma-262.htm.
5. P. Niemeyer. BeanShell 2.0 Documentation, http://www.beanshell.org/.
6. Sun Microsystems, Inc. Java Speech Grammar Format specification. Available at http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/, 1998.
7. He, Y., Young, S.: Semantic processing using the Hidden Vector State model. Computer Speech and Language, Volume 19, Issue 1, 2005, 85-106.
8. Hošna, M.; Konopík, M.: Voice Controlled Chess usable also for Blind People . In SPECOM 2007 Proceedings. Moskva : Moscow State Linquistic University , 2007. s. 725-728. ISBN 6-7452-0110-X.